C++ Standard Template Library (STL)

IF2210 – Semester II 2022/2023

Motivasi

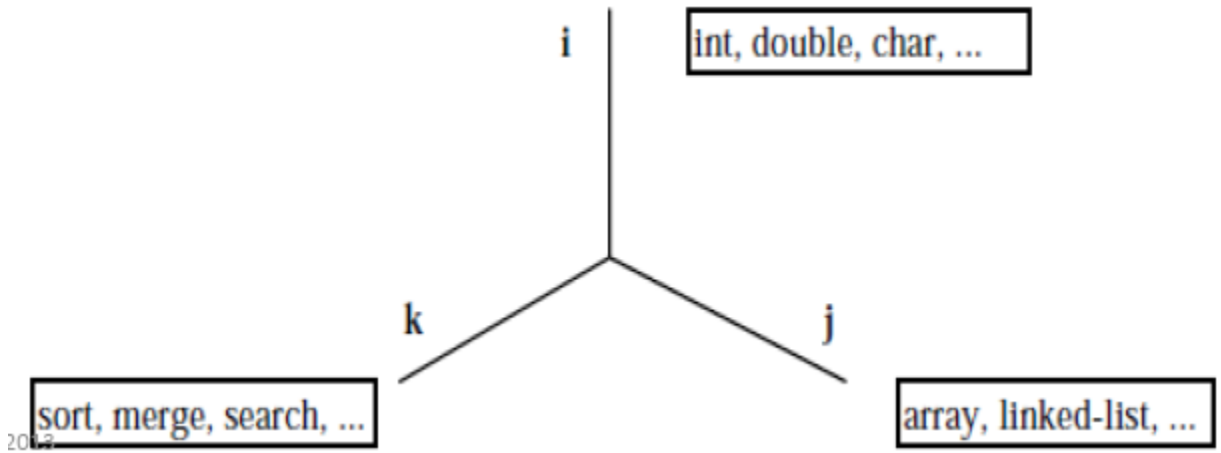
- Alexander Stepanov (1970an):
  - *“some algorithms do not depend on some particular implementation of a data structure, but only a few fundamental semantic properties of the structure”*
  - *“fundamental semantic properties of the structure”*: e.g. how to get one element, how to get the next, how to step through the beginning to the end, …

Sejarah

- 1985: Stepanov built generic Ada library, and was asked if he could do in C++ as well
- 1987: Template has yet implemented in C++
- 1988: Stepanov moved to the HP Labs
- 1992: Stepanov was appointed as manager of an algorithm projects:
  - He and Meng Lee wrote STL, to build algorithms defined as generically as possible without losing efficiency

Intro to STL

- STL (Standard Template Library) is a component library, described in a clean and formally sound concepts
- The idea is “the orthogonal decomposition of the component space”



STL Components

- Containers
  - Template of data structures
  - Object that can keep and administer objects
- Iterators
  - Like pointers, access elements of containers
- Algorithms
  - Computational processor that can work on different containers

Containers

Intro to Containers

- Tiga jenis *container*:
  - *Sequence container*
    - Struktur data linear (*vector*, *linked list*)
    - *First-class container*
  - *Associative container*
    - Tidak linear, pencarian elemen lebih cepat
    - Pasangan *key/value*
    - *First-class container*
  - *Container adapter*
- *Near/partial container*: mirip *container*, dengan fungsionalitas terbatas
- *Container* memiliki fungsi-fungsi yang sama (*common*)

Kelas-kelas Container pada STL

- *Sequence container*:
  - `vector`, `deque`, `list`
- *Associative container*:
  - `set`, `multiset`, `map`, `multimap`
- *Container adapter*:
  - `stack`, `queue`, `priority_queue`

Fungsi anggota STL

- Fungsi anggota untuk semua *container*:
  - ctor, cctor, dtor
  - `empty`, `max_size`, `size`, `= < <= > >= == !=`, `swap`
- Fungsi untuk *first-class container*:
  - `begin`, `end`
  - `rbegin`, `rend`
  - `erase`, `clear`

typedef umum di STL

- `typedef` untuk *first-class container*:
  - `value_type`
  - `reference`
  - `const_reference`
  - `pointer`
  - `iterator`
  - `const_iterator`
  - `reverse_iterator`
  - `const_reverse_iterator`
  - `difference_type`
  - `size_type`

Iterators

Intro to Iterators (1)

- *Iterator* mirip dengan *pointer*
  - Menunjuk ke elemen pertama sebuah *container*
  - Operator untuk iterator sama pada semua *container*:
    - * *dereference*
    - `++` menunjuk ke elemen berikutnya
    - `begin()` mengembalikan *iterator* ke elemen pertama
    - `end()` mengembalikan *iterator* ke elemen terakhir
  - Iterator digunakan terhadap sekuens (rentang/*ranges*):
    - *Containers*
    - *Input sequences*: `istream_iterator`
    - *Output sequences*: `ostream_iterator`

Intro to Iterators (2)

- Penggunaan:
  - `std::istream_iterator <int> inputInt(cin)`
    - Membaca input dari `cin`
    - `*inputInt`: *dereference* ke `int` pertama dari `cin`
    - `++inputInt`: pindah ke `int` berikutnya pada *stream*
  - `std::ostream_iterator <int> outputInt(cout)`
    - Menulis `int` ke `cout`
    - `*outputInt = 7`: menulis 7 ke `cout`
    - `++outputInt`: memajukan *iterator* supaya dapat menulis `int` berikutnya

Outline

fig21_05.cpp
(1 of 2)

```
// Fig. 21.5: fig21_05.cpp
// Demonstrating input and output with iterators.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <iterator>  // ostre                    ator

int main()
{
   cout << "Enter two integer

   // create istream_iterator for reading int values from cin
   std::istream_iterator< int > inputInt( cin );

   int number1 = *inputInt;  // read int from standard input
   ++inputInt;               // move iterator to next input value
   int number2 = *inputInt;  // read int from standard input

```

Note creation of **`istream_iterator`**. For compilation reasons, we use **`std::`** rather than a **`using`** statement.

Access and assign the iterator like a pointer.

```
   // create ostream_iterator for writing int values to cout
   std::ostream_iterator< int > outputInt( cout );

   cout << "The sum is: ";
   *outputInt = number1 + number2;  // output result to cout
   cout << endl;

   return 0;

} // end main
```

```
Enter two integers: 12 25
The sum is: 37
```

Create an **ostream_iterator** is similar. Assigning to this iterator outputs to **cout**.

Outline

fig21_05.cpp (2 of 2)

fig21_05.cpp output (1 of 1)

Jenis-jenis iterator

- **Input:** membaca elemen dari *container*, hanya bisa maju
- **Output:** menulis elemen ke *container*, hanya bisa maju
- **Forward:**
  - gabungan *input* dan *output*, mempertahankan posisi
  - *multi-pass* (dapat melewati sekuens dua kali)
- **Bidirectional:** seperti *forward*, tapi bisa mundur juga
- **Random access:** seperti *bidirectional*, tapi bisa lompat ke elemen manapun

Jenis iterator yang didukung container

- *Sequence container*
  - `vector`, `deque`: random access
  - `list`: bidirectional
- *Associative container*
  - `set`, `multiset`, `map`, `multimap`: bidirectional
- *Container adapter*
  - `stack`, `queue`, `priority_queue`: tidak mendukung iterator

Operasi pada iterator (1)

- Semua:
  - `++p, p++`
- Iterator input:
  - `*p`
  - `p = p1`
  - `p == p1, p != p1`
- Iterator output:
  - `*p`
  - `p = p1`
- Iterator forward:
  - Memiliki fungsionalitas iterator input dan output

Operasi pada iterator (2)

- Iterator bidirectional:
  - `--p`, `p--`
- Iterator random access:
  - `p+i`, `p+=i`
  - `p-i`, `p-=i`
  - `p[i]`
  - `p<p1`, `p<=p1`
  - `p>p1`, `p>=p1`

Algorithms

Intro to Algorithms

- STL memiliki algoritma-algoritma yang digunakan secara generik untuk setiap *container*
  - Beroperasi terhadap elemen (secara tidak langsung, melalui iterator)
  - Beroperasi pada sekuens elemen
    - Didefinisikan oleh pasangan iterator (elemen pertama dan terakhir)
  - Algoritma biasanya mengembalikan iterator
    - Contoh: `find()` mengembalikan iterator yg menunjuk ke elemen yang dicari, atau mengembalikan `end()` jika tidak ketemu
  - Algoritma *premade* menghemat waktu & usaha pemrogram

Algoritma

- Sebelum STL
  - *Library* kelas tidak saling kompatibel
  - Algoritma tertanam di kelas-kelas container
- STL memisahkan algoritma dari *container*
  - Lebih mudah untuk menambah algoritma baru
  - Lebih efisien, menghindari pemanggilan fungsi `virtual`
  - `<algorithm>`

Algoritma dasar *searching* & *sorting*

- `find(iter1, iter2, value)`: mengembalikan iterator ke kemunculan pertama `value` pada rentang `iter1` sampai sebelum `iter2`
- `find_if(iter1, iter2, function)`: seperti `find`, tapi mengembalikan iterator ketika `function` mengembalikan `true`
- `sort(iter1, iter2)`: mengurutkan elemen secara menaik (*ascending*)
- `binary_search(iter1, iter2, value)`: mencari elemen pada sekuens yang terurut menaik, dengan algoritma pencarian biner

Outline

fig21_31.cpp
(1 of 4)

```
// Fig. 21.31: fig21_31.cpp
// Standard library search and sort algorithms.
#include <iostream>

using std::cout;
using std::endl;

#include <algorithm>  // algorithm definitions
#include <vector>     // vector class-template definition

bool greater10( int value );  // prototype

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };

   std::vector< int > v( a, a + SIZE );
   std::ostream_iterator< int > output( cout, " " );

   cout << "Vector v contains: ";
   std::copy( v.begin(), v.end(), output );

   // locate first occurrence of 16 in v
   std::vector< int >::iterator location;
   location = std::find( v.begin(), v.end(), 16 );
```

Outline

fig21_31.cpp
(2 of 4)

```
   if ( location != v.end() )
      cout << "\n\nFound 16 at location "
           << ( location - v.begin() );
   else
      cout << "\n\n16 not found";

   // locate first occurrence of 100 in v
   location = std::find( v.begin(), v.end(), 100 );

   if ( location != v.end() )
      cout << "\nFound 100 at location "
           << ( location - v.begin() );
   else
      cout << "\n100 not found";

   // locate first occurrence of value greater than 10 in v
   location = std::find_if( v.begin(), v.end(), greater10 );

   if ( location != v.end() )
      cout << "\n\nThe first value greater than 10 is "
           << *location << "\nfound at location "
           << ( location - v.begin() );
   else
      cout << "\n\nNo values greater than 10 were found";

```

```
   // sort elements of v
   std::sort( v.begin(), v.end() );

   cout << "\n\nVector v after sort: ";
   std::copy( v.begin(), v.end(), output );

   // use binary_search to locate 13 in v
   if ( std::binary_search( v.begin(), v.end(), 13 ) )
      cout << "\n\n13 was found in v";
   else
      cout << "\n\n13 was not found in v";

   // use binary_search to locate 100 in v
   if ( std::binary_search( v.begin(), v.end(), 100 ) )
      cout << "\n100 was found in v";
   else
      cout << "\n100 was not found in v";

   cout << endl;

   return 0;

} // end main
```

Outline

fig21_31.cpp
(3 of 4)

```
// determine whether argument is greater than 10
bool greater10( int value )
{
   return value > 10;

} // end function greater10
```

```
Vector v contains: 10 2 17 5 16 8 13 11 20 7

Found 16 at location 4
100 not found

The first value greater than 10 is 17
found at location 2

Vector v after sort: 2 5 7 8 10 11 13 16 17 20

13 was found in v
100 was not found in v
```

Outline

fig21_31.cpp (4 of 4)

fig21_31.cpp output (1 of 1)

Contoh-contoh: `vector` dan `stack`

Sequence Container

- Ada tiga *sequence container*
  - `vector` – berbasis *array*
  - `deque` – berbasis *array*
  - `list` – *linked list* yang *robust*

We will only discuss vector

vector (1)

- `vector`
  - **Header** `<vector>`
  - Struktur data dengan lokasi memori kontigu
    - Akses elemen dengan [ ]
  - Digunakan jika data harus diurutkan dan data harus mudah diakses
- Ketika memori yang teralokasi penuh:
  - Alokasikan area memori kontigu yang lebih besar
  - Salin isi ke area memori baru tsb
  - Dealokasi memori yang lama
- Memiliki iterator *random access*

vector (2)

- Deklarasi:
  - `std::vector <type> v;`
    - *type*: `int`, `float`, etc.
  - Iterator:
    - `std::vector<type>::const_iterator iterVar;`
      - tidak dapat memodifikasi elemen
    - `std::vector<type>::reverse_iterator iterVar;`
      - Iterasi elemen dari belakang (mundur)
      - Starting point: `rbegin`
      - Ending point: `rend`

Fungsi-fungsi pada vector (1)

- `v.push_back(value)`: menambah elemen di akhir *vector* (dimiliki oleh semua *container* sekuens)
- `v.size()`: ukuran *vector* saat ini
- `v.capacity()`: jumlah elemen yang dapat ditampung sebelum realokasi. Realokasi menggandakan ukuran
- `vector<`*`type`*`> v(a, a+SIZE)`: membuat *vector* `v` dengan elemen dari *array* `a` sebanyak `SIZE`

Fungsi-fungsi pada vector (2)

- `v.insert(iterator, value)`: menambahkan elemen `value` di depan lokasi `iterator`
- `v.insert(iterator, array, array+SIZE)`: menambahkan elemen `array` sejumlah `SIZE` ke `v`
- `v.erase(iterator)`: hapus elemen dari *container*
- `v.erase(iter1, iter2)`: hapus elemen pada `iter1` hingga sebelum `iter2`
- `v.clear()`: kosongkan *container*

Fungsi-fungsi pada vector (3)

- `v.front()`, `v.back()`: mengembalikan elemen pertama dan terakhir
- `v[elementNumber] = value;` : meng-*assign* `value` ke sebuah elemen
- `v.at(elementNumber) = value;` : sama dengan sebelumnya, tapi dengan pemeriksaan indeks. Melempar *exception* `out_of_bounds`

Iterator ostream

- `std::ostream_iterator <type> Name(outputStream, separator);`
  - *type*: jenis tipe data yang dikeluarkan
  - `outputStream`: iterator lokasi keluaran
  - `separator`: karakter yang memisahkan keluaran
- Contoh:
  - `std::ostream_iterator <int> output(cout, " ");`
  - `std::copy(iter1, iter2, output);`
    - Menyalin elemen dari posisi `iter1` sampai sebelum `iter2` ke output, sebuah `ostream_iterator`

Outline

fig21_14.cpp
(1 of 3)

```
// Fig. 21.14: fig21_14.cpp
// Demonstrating standard library vector class template.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <vector>  // vector class-template definition

// prototype for function template printVector
template < class T >
void printVector( const std::vector< T > &integers2 );

int main()
{
   const int SIZE = 6;
   int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 };

   std::vector< int > integers;

   cout << "The initial size of integers is: "
        << integers.size()
        << "\nThe initial capacity of integers is: "
        << integers.capacity();

```

Create a **vector** of **int**s.

Call member functions.

```
   // function push_back is in every sequence collection
   integers.push_back( 2 );
   integers.push_back( 3 );
   integers.push_back( 4 );

   cout << "\nThe size of integers is: " << integers.size()
        << "\nThe capacity of integers is: "
        << integers.capacity();

   cout << "\n\nOutput array using pointer notation: ";

   for ( int *ptr = array; ptr != array + SIZE; ++ptr )
      cout << *ptr << ' ';

   cout << "\nOutput vector using iterator notation: ";
   printVector( integers );

   cout << "\nReversed contents of vector integers: ";

```

Add elements to end of **vector** using **push_back**.

fig21_14.cpp
(2 of 3)

Outline

```
   std::vector< int >::reverse_iterator reverseIterator;

   for ( reverseIterator = integers.rbegin();
         reverseIterator!= integers.rend();
         ++reverseIterator )
      cout << *reverseIterator << ' ';

   cout << endl;

   return 0;

} // end main

// function template for outputting vector elements
template < class T >
void printVector( const std::vector< T > &integers2 )
{
   std::vector< T >::const_iterator constIterator;

   for ( constIterator = integers2.begin();
         constIterator != integers2.end();
         constIterator++ )
      cout << *constIterator << ' ';

} // end function printVector
```

Walk through **vector** backwards using a **reverse_iterator**.

Template function to walk through **vector** forwards.

```
The initial size of v is: 0
The initial capacity of v is: 0
The size of v is: 3
The capacity of v is: 4

Contents of array a using pointer notation: 1 2 3 4 5 6
Contents of vector v using iterator notation: 2 3 4
Reversed contents of vector v: 4 3 2
```

Outline

fig21_14.cpp output (1 of 1)

Outline

fig21_15.cpp
(1 of 3)

```
// Fig. 21.15: fig21_15.cpp
// Testing Standard Library vector class template
// element-manipulation functions.
#include <iostream>

using std::cout;
using std::endl;

#include <vector>     // vector class-template definition
#include <algorithm>  // copy algorithm

int main()
{
   const int SIZE = 6;
   int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 };

   std::vector< int > integers( array, array + SIZE );
   std::ostream_iterator< int > output( cout, " " );

   cout << "Vector integers contains: ";
   std::copy( integers.begin(), integers.end(), output );

   cout << "\nFirst element of integers: " << integers.front()
        << "\nLast element of integers: " << integers.back();

```

Create **vector** (initialized using an array) and **ostream_iterator**.

Copy range of iterators to **output** (**ostream_iterator**).

Outline

```
   integers[ 0 ] = 7;       // set first element to 7
   integers.at( 2 ) = 10;   // set element at position 2 to 10

   // insert 22 as 2nd element
   integers.insert( integers.begin() + 1, 22 );

   cout << "\n\nContents of vector integers after changes: ";
   std::copy( integers.begin(), integers.end(), output );

   // access out-of-range element
   try {
      integers.at( 100 ) = 777;

   } // end try

   // catch out_of_range exception
   catch ( std::out_of_range outOfRange ) {
      cout << "\n\nException: " << outOfRange.what();

   } // end catch

   // erase first element
   integers.erase( integers.begin() );
   cout << "\n\nVector integers after erasing first element: ";
   std::copy( integers.begin(), integers.end(), output );

```

More **vector** member functions.

at has range checking, and can throw an exception.

```
   // erase remaining elements
   integers.erase( integers.begin(), integers.end() );
   cout << "\nAfter erasing all elements, vector integers "
        << ( integers.empty() ? "is" : "is not" ) << " empty";

   // insert elements from array
   integers.insert( integers.begin(), array, array + SIZE );
   cout << "\n\nContents of vector integers before clear: ";
   std::copy( integers.begin(), integers.end(), output );

   // empty integers; clear calls erase to empty a collection
   integers.clear();
   cout << "\nAfter clear, vector integers "
        << ( integers.empty() ? "is" : "is not" ) << " empty";

   cout << endl;

   return 0;

} // end main
```

Outline

fig21_15.cpp
(3 of 3)

```
Vector integers contains: 1 2 3 4 5 6
First element of integers: 1
Last element of integers: 6

Contents of vector integers after changes: 7 22 2 10 4 5 6

Exception: invalid vector<T> subscript

Vector integers after erasing first element: 22 2 10 4 5 6
After erasing all elements, vector integers is empty

Contents of vector integers before clear: 1 2 3 4 5 6
After clear, vector integers is empty
```

Outline

fig21_15.cpp output (1 of 1)

```
#include <iostream>
using namespace std;
#include <vector>
#include <stdio.h>

int main() {
  vector<int> v;
  vector<int>::const_iterator CI;
  vector<int>::reverse_iterator RI;

  /* push five elements into the vector */
  v.push_back(1);
  v.push_back(2);
  v.push_back(3);
  v.push_back(4);
  v.push_back(5);
  cout << "Size of vector: " << v.size() << endl;
  cout << "Initial capacity vector: " << v.capacity() << endl;

  /* print the vector */
  for (CI = v.begin(); CI != v.end(); CI++) {
    cout << *CI << " ";
  }
  cout << endl;

  /* print the vector backward */
  for (RI = v.rbegin(); RI != v.rend(); RI++) {
    cout << *RI << " ";
  }

  return 0;
}
```

Container Adapter

- *Container adapter*
  - `stack`, `queue`, dan `priority_queue`
  - Bukan *first-class container*
    - Tidak mendukung iterator
    - Tidak menyediakan struktur data yang sebenarnya
  - Pemrogram dapat memilih implementasi yang diinginkan
  - Fungsi anggota: `push` dan `pop`

We will discuss only stack

stack

- `stack`
  - Header `<stack>`
  - Tambah & hapus data dari salah satu ujung saja: LIFO
  - Secara internal dapat menggunakan `vector`, `list`, atau `deque` (*default*)
  - Deklarasi:

```
stack <type, vector<type>> myStack;
stack <type, list<type>> myOtherStack;
stack <type> anotherStack; // default deque
```

    - struktur data internal tidak mempengaruhi *behavior*, hanya kinerja (`deque` dan `vector` lebih cepat daripada `list`)

Outline

fig21_23.cpp
(1 of 3)

```cpp
// Fig. 21.23: fig21_23.cpp
// Standard library adapter stack test program.
#include <iostream>

using std::cout;
using std::endl;

#include <stack>   // stack adapter definition
#include <vector>  // vector class-template definition
#include <list>    // list class-template definition

// popElements function-template prototype
template< class T >
void popElements( T &stackRef );

int main()
{
   // stack with default underlying deque
   std::stack< int > intDequeStack;

   // stack with underlying vector
   std::stack< int, std::vector< int > > intVectorStack;

   // stack with underlying list
   std::stack< int, std::list< int > > intListStack;

```

Create stacks with various implementations.

Outline

```
   // push the values 0-9 onto each stack
   for ( int i = 0; i < 10; ++i ) {
      intDequeStack.push( i );
      intVectorStack.push( i );
      intListStack.push( i );

   } // end for

   // display and remove elements from each stack
   cout << "Popping from intDequeStack: ";
   popElements( intDequeStack );
   cout << "\nPopping from intVectorStack: ";
   popElements( intVectorStack );
   cout << "\nPopping from intListStack: ";
   popElements( intListStack );

   cout << endl;

   return 0;

} // end main
```

Use member function **push**.

21_23.cpp
(2 of 3)

```
// pop elements from stack object to which stackRef refers
template< class T >
void popElements( T &stackRef )
{
   while ( !stackRef.empty() ) {
      cout << stackRef.top() << ' ';  // view top element
      stackRef.pop();                 // remove top element

   } // end while

} // end function popElements
```

```
Popping from intDequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from intVectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from intListStack: 9 8 7 6 5 4 3 2 1 0
```


fig21_23.cpp (3 of 3)

fig21_23.cpp output (1 of 1)

```
#include <iostream>
using namespace std;
#include <stack>
#include <stdio.h>

int main() {
  stack<int> st;
  /* push three elements into the stack */
  st.push(1);
  st.push(2);
  st.push(3);
  /* pop & print 2 elements from the stack */
  cout << st.top() << " ";
  st.pop();
  cout << st.top() << " ";
  st.pop();
  /* modify top element */
  st.top() = 77;
  /* push two new elements */
  st.push(4);
  st.push(5);
  /* pop 1 element without processing it */
  st.pop();
  /* pop and print remaining elements */
  while (!st.empty()) {
    cout << st.top() << " ";
    st.pop();
  }
  cout << endl;
  return 0;
}
```

References

- H.M. Deitel, P.J. Deitel: *"How to Program in C++"*, Prentice Hall (...)
- http://en.cppreference.com/w/cpp/container